PROG 3

Intro

C++ offers many language features, such as

- Procedural programming
- Object-oriented programming
- Generic meta programming
- Functional programming

Large code bases can be handled, C++ allows easy access to C APIs, allows low level optimizations, and is a very powerful language.

Hello World

Standard libary elements are in the **namespace** "std" and can be accessed with the **#include** keyword.
C++ compilers generate platform dependent binaries, f.e. Java is platform independent. C++ programs need to be compiled for each platform.

How to edit & compile

Edit a C++ file wit `gedit helloworld.cpp` and compile it with `g++ -c helloworld.cpp` .
Link the file and build an executable with `g++ helloworld.o -o helloworld.exe` . –> `./helloworld.exe`

Declarations & Definitions

Declarations introduce the existence of structures, variables, functions, etc. –> declare a function:

```
int add(int, int);
```

Definitions are declarations, which contain all information about the declared thing. –> define a function:

```
int add(int a, int b) {
    return a + b;
}
```

ODR (One Definition Rule)

- Only one single definition of a function, variable, class, etc. is allowed.
- Every used thing must be defined somewhere.

Redeclaration of a function is allowed, if the definition is the same.

Modularization

Header files contain declarations, which can be included in other files.

```
// helloworld.h
#ifndef HELLOWORLD_H
#define HELLOWORLD_H
    int add(int, int);
#endif
```

```
// helloworld.cpp
#include "helloworld.h"
    int add(int a, int b) {
        return a + b;
    }
```

Libaries are collections of header files, which can be included in other files. They can be either static (*.a*/.LIB) or dynamic (*.so*/.DLL).

Namespaces

Namespaces are used to avoid name collisions.

```
namespace mynamespace {
    int add(int a, int b) {
        return a + b;
    }
}

int main() {
    int a = 1;
    int b = 2;
    int c = mynamespace::add(a, b);
    return 0;
}
```

Makefiles

Makefiles are used to automate the build process.

CMake is a Makefile generator, which can be used to generate Makefiles for different platforms.

First Steps

Functions

- Functions can be defined for different types. –> **overloading**
- Function calls with ambiguous types are not allowed. –> **overloading resolution**

Variables, Narrowing

- Variables are defined prior usage.
- Initialization `a=2` is deprecated, use `a{2}` or `a={2}` instead.
- Narrowing: losing information during type conversion. –> `int a = 2.5;`
- Array variables are defined: `TYPE arr[NUM]`.
- C++11 defined std::array `std::array<TYPE, NUM> arr`.
- Array sizes must be known at compile time.

Constants

- Constants are defined with `const`.
- const variabes protect variables from modification.
- constexpr variables protect variables from modification and allow compile time evaluation.

Refrences & Pointers

Pointers Features:

- Pointer = address (where) + optional: type (what)
- Nullpointer = `nullptr`
- Pointer arithmetic: address modifications

Use cases:

- Data structures –> Lists
- Data referencing (passing pointers instead of values)
- Dynamic memory management

Pointer declaration `TYPE* name {...};`
Addresses of variables can be accessed with `&name`
Pointer arithmetic is possible –> `&c2-&c1`
To access the data to which a pointer is pointing use the dereference operator `*` –> `*name=2;`

```
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
int main() {
    int x{2},y{3};
    int *xp = &x;
    swap(xp, &y);
    swap(&x, &y);
}
```

References

- Reference variable declaration: `TYPE& name{...};` - no reassignment possible
- References are aliases for variables
- Accessing a reference is the same as accessing the original value
- References can't be null

C-Strings

- C-Strings are arrays of characters,
- `const TYPE* ptr`is a pointer to a const TYPE
- `TYPE* const ptr`is a const pointer to a TYPE

Different function parameters **Pass by value** `func (TYPE value)` –> copy value (input, small TYPES)
Pass by reference:
`func (TYPE &value)` –> reference to original value (input, output) `func (const TYPE &value)` –> reference to original value (input, large TYPES)
Pass by pointer:
`func (TYPE *value)` –> reference to original value (input, output)
`func (const TYPE *value)` –> reference to original value (input)

Dynamic Memory Management

Allocation `TYPE* ptr = new TYPE{init};`
Deallocation `delete ptr;`
Allocate N data element `TYPE* ptr = new TYPE[N];`
Access `ptr[i]`
Deallocate `delete[] ptr;`
Dangling pointer is a pointer, which points to a deallocated memory location.

I/O

Open a file `std::ifstream`
Read from a file `std::getline(std::cin, line);`
Write to a file `std::cout << "Hello World" << std::endl;`

Classes & Objects

Classes

- Classes are identified with the keyword `class` or `struct`.
- Member variables are defined in the class.
- The constructor has the same name as the class and is called when an object is created.
- The destructor has the same name as the class following a tilde, has no parameters and is called when an object is destroyed.

```
class MyClass {
    public:
        MyClass(int a, int b);
        ~MyClass() {}
        int compete();
    private:
        int a, b;
};
```

- Access modifiers control how members can be accessed.
  - `public` accessible from everywhere, default for structs
  - `private`accessible from inside only, default for classes
  - `protected` accessible from inside and subclasses

Objects

Syntax: `ClassName variableName;` or `ClassName variableName{...};` or `ClassName variableName(...);` or `ClassName variableName{};`

Storage Duration

Static storage duration is the lifetime of a variable, which is the whole program.
Automatic storage duration –> local variables, initialized when entering the scope and destroyed when leaving the scope.
Dynamic storage duration –> user controlled lifetime, allocated with `new` and deallocated with `delete`.

Modern Storage Duration

Rule: do not use new/delete in modern C++.

- `std::shared_ptr<TYPE>` is a smart pointer, which manages the lifetime of an object.
- `std::unique_ptr<TYPE>` is a smart pointer, which manages the lifetime of an object and can't be copied/shared.

- `std::weak_ptr<TYPE>` is a smart pointer without ownership - must be converted to a shared_ptr to access the object.
- No need to delete objects, which are managed by smart pointers.

Rule: use raw pointers with care in modern C++.

- Use **shared_ptr** instead of **T*** to express shared ownership.
- Use **unique_ptr** instead of **T*** to express private ownership.
- Use **weak_ptr** instead of **T*** to express no ownership.

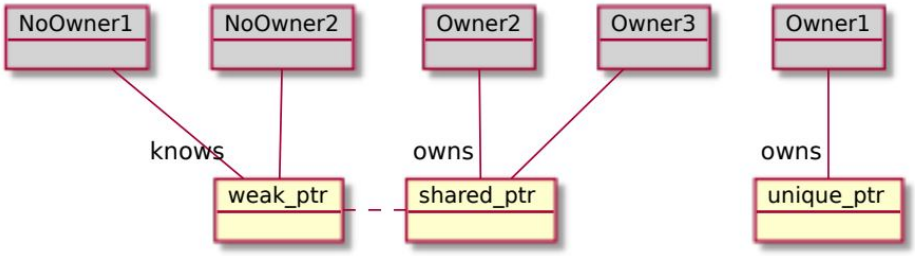

Figure 1: Smart Pointers

Inline Functions

Implicit inline member functions are functions, which are defined in the class declaration.
Explicit inline (member) functions start their definition with the keyword `inline`.
constexpr functions are implicitly inline.

Const Methods

Syntax: add `const` after the parameter list.
Compiler guarantees that the method does not modify the object. In a const context the compiler only allows const access.
`const` allows to control whether a function is allowed to be called or not in a given context.

Constexpr Functions

Syntax: add `constexpr` in front of the function name.
Semantic:
constexpr functions are enabled to be used in constexpr expressions.
constexpr functions may only use restricted language features.
Notes:
constexpr constructors are possible
a constexpr function can be called at compile time or at runtime

Static Members

Syntax: add `static` in front of the member declaration.
They exist once per class and have static storage duration. - Definition and initialization happens outside the class

Unions

Unions are data structures, which can store different types of data in the same memory location.
Unions can only store one attribute at a time. The programmer is responsible for managing what attribute is stored in the union.
`std::variant` are modern unions and manage themselves what attribute is stored.